

Machine Learning Project 1

To: Doctor Karlsson
From: Micah Runner (Undergraduate)
Subject: Machine Learning Project 1: Perceptron
Date: September 25, 2020

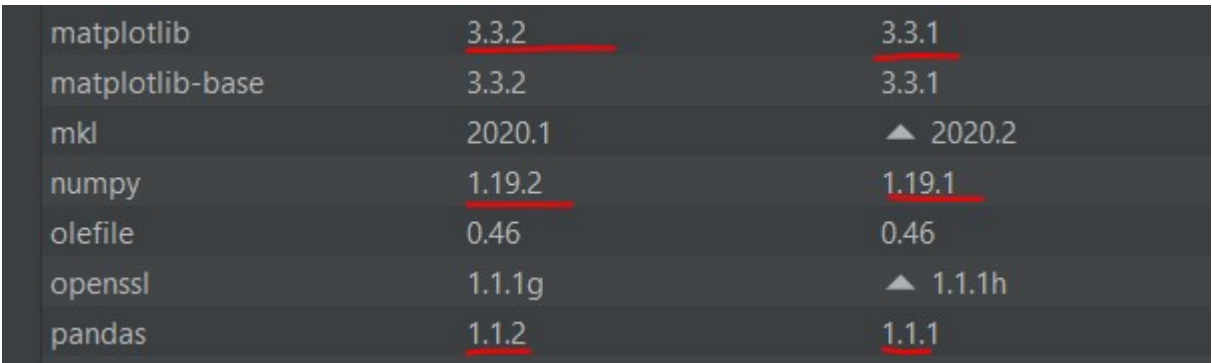
Introduction

The point of this project was to introduce students to the Perceptron Class, which is a form of machine learning. The Perceptron Class is a supervised learner, as in it uses a set of biases (weights) to work itself into the right answer, or within a certain amount error. The Perceptron Class works by taking a data and that has been split into a 0 or 1, or binary set, as well as a training set of data. It uses a linear function and its weights to slowly learn how to predict an output.

Software Minimum Requirements

Python – Version 3.8.5
matplotlib – Version 3.3.2 (IDE says 3.3.2, but also says latest version is 3.3.1, so it must be from the future)
numpy – Version 1.19.2 (IDE says 1.19.2, but it says the latest version is 1.19.1, so it must be from the future)
pandas – Version 1.1.2 (IDE says 1.1.2, but it says the latest version is 1.1.1, so again it must be from the future)

We used the *Miniconda3* environment using the *PyCharm* IDE to create and test the Class. The figure below shows the time traveling libraries being used.



matplotlib	<u>3.3.2</u>	<u>3.3.1</u>
matplotlib-base	3.3.2	3.3.1
mkl	2020.1	▲ 2020.2
numpy	<u>1.19.2</u>	<u>1.19.1</u>
olefile	0.46	0.46
openssl	1.1.1g	▲ 1.1.1h
pandas	<u>1.1.2</u>	<u>1.1.1</u>

Figure 1: Image of the *PyCharm* IDE showing the library version currently used, which is the left list, and the right list is the latest version.

Implementation

Here is what the instructor provided to the students and what the program requirements.

Programming Assignment:

This is the first part of your machine learning library. We will focus on the perceptron.

1. Create the perceptron class:
 - a. A new perceptron instance should be created with a rate and the numbers of iterations.
 - b. It should be generic enough that it can take a training set of any size. My write-up only uses two features, but my code can take any number of features.
 - c. The class has four functions:
 - i. `__init__()`
 - ii. `fit()`
 - iii. `net_input()`
 - iv. `predict()`
2. The example in the text always runs the provided number of iterations, even if it has converged long before that. Fix that problem (i.e. in the text case it should run only 6 iterations).
3. Fix and put the `plot_decision_regions` into your ML-library, so you can use it.
4. Try your implementation on other pairs, and other triples of the Iris-data set.

The professor provided an outline of the entire project to the undergraduate students, which meant that as long as one understood *Python*, it would be easy. Unfortunately, this was the first time we have used *Python*, so this required a lot more Googling than other projects. Specifically, learning about the use of the *Numpy* library, which provides a lot of matrix functions. This means that we can finally read off a mathematician's ramblings and turn it into something useful. Students were also recommended to use `matplotlib`, which we think is what *Matlab* runs off of, so some of the functions seemed more natural to me.

We did not have to change anything within the `__init__()` function as the professor provided the entire thing to us, we did however add comments about what each variable accomplishes for the class. This function creates the class itself and is where the user can pass in training rate as well as how many iterations it will run and hopefully learn its purpose in life. Moving on to the next required function.

The `fit()` function is where the heavy lifting takes place within the Perceptron Class. This function takes in a training vector that is composed of samples and features. The samples is just raw training data that it will use to work on improving itself. The features is how many different types of training is within the data set. Lastly, this function takes in a list of target values, these are the correct responses to the data set that the Perceptron Class should achieve.

The Perceptron Class uses an array, specially a *ndarray* from the *Numpy* library, whose size is that of the amount of features plus one, which is due to how the algorithm works. The first index is the Basis of the entire Perceptron, the other two weights are meant to help the Perceptron bring balance to the world. One could hypothetically create multiple instances of the Perceptron Class and then feed their outputs into the inputs of another layer of Perceptron's, but such an idea would never work... Anyway, some people prefer to use a completely random bias value, but we did not do that for our class. We decided to loop through the training data collecting any mistakes it made into a list that holds the amount of errors the Perceptron Class

makes on its way to learning the training data. As the Perceptron is biased by the learning rate times the delta between the target data and it's attempt at predicting the rate, this value will be stored in the bias index of the weight array, but this value will also be used for the weights themselves.

The Perceptron Class will form its weights, or inputs, by taking the bias and multiplying it by the current raw data itself and add itself its previous value. If the bias is not equal to zero, then we start a counter and add one to itself. After going through all of the data we will then add the error to the error list. Basically this is how computer scientists treat machines as children, this method is the same as watching a kid fall down a bunch times while trying to ride a bike until they figure it out. Of course, we do not want a kid to keep proving a million times that they know how to ride a bike on their own, so we also need to check and see if the Perceptron has mastered the data set. To solve this problem we decided that the Perceptron Class has mastered the data set if it has two runs with a bias change of zero. If this is the case then we return the fully trained Perceptron Class.

The **predict()** function was provided by the professor, but it works by calling the **net_input()** function given the training data, then determines if that result is greater than equal to zero. It then returns one if it is true or it selects the negative one if the expression is false. The **net_input()** function utilizes the magic of the *Numpy* library and the power of a dot product. This function simply takes in part of the data set, then performs the dot product between the data and its weights. It then adds this result with bias. There is probably more complicated way of doing this, but *Numpy* is a great library to utilize, so this was as simple as following the Professor's comment on what to return.

Lastly, we had to fix the **plot_decision_regions()** function, which takes in a 2d data set, a classifier, and a resolution value. It then takes in data and finds minimums and maximums to create the 2d grid, which helps with viewing the data set in a nice viewing plot. We then figure out what the function output the classifier takes the shape of and then run the contour function. This shades in the regions between what the classifier has determined separates the data. We then loop through the data sets themselves and assign a color and shape value on our 2d plot. Lastly, we added in a legend, so the user can see what values their points have been given. The professor gave us "broken code" that still functioned but gave out an error traced to *matplotlib* version 3.0, which we learned through trying to Google the error message. The easiest solution is a classic programming, which is to pretend that it does not exist. This work around was given by user *Max Kleiner* from stack overflow, here is the link to the thread of their recommendation to this error <https://stackoverflow.com/questions/55109716/c-argument-looks-like-a-single-numeric-rgb-or-rgba-sequence>. However, we can only show a plot of a two-feature set. We believe that this is what the professor wanted, but given more time, we should be able to figure out a way of making it work on any given feature size or at least limited by the number of colors that are supported by the library.

Running the Perceptron Class

To run the Perceptron Class, the user must download the ML.py file into a folder that they are planning on using the Perceptron Class. The user must then provide a training vector X, whose shape takes the form of [number of samples, number of features], a target vector Y, whose values must either be -1 or 1, and be the same length as the number of samples within the X vector. For our first example we will use the *Iris* data set, given the following URL link 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'.

We will import the following libraries *pandas*, *numpy*, and *matplotlib*. The snippet below shows the following *python* console, assigning the libraries to an object.

```
>>> import pandas as pd, numpy as np, matplotlib.pyplot as plt
```

Figure 2: Importing the needed libraries to run the Perceptron Class.

Next we will gather the data using the *pandas* library function *read_csv()*, which allows the data set to be read from a csv file. We then create a list from the 4th column of the *Iris* data set, by using **y = df.iloc[0:150, 4].values**. We then need to change the values of the desired output, which in this case is the y list values that match the *Iris-setosa* label and assign it -1 otherwise it becomes a 1, by using **y = np.where(y == 'Iris-setosa', -1, 1)**. Lastly, we need the raw data input, which we will grab from the zero column and the second column of the data set by using **X = df.iloc[0:150, [0,2]].values**. All of this is shown in the figure below.

```
.. df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data', header=None)
.. y = df.iloc[0:150, 4].values
.. y = np.where(y == 'Iris-setosa', -1, 1)
.. X = df.iloc[0:150, [0,2]].values
```

Figure 3: Creating the needed vectors using the *Iris* dataset.

Now we will need to import from the *ML.py* file in which the Perceptron Class exists by using **from ML import Perceptron**. We will then create an object to become the almighty binary overlord that the Perceptron Class is by using **pn = Perceptron(0.1, 10)**. As stated before the values of the Perceptron are as the learning rate and number of iterations that the Perceptron will use to learn the dataset. Lastly, we will use the object and train it on the data sets by using **pn.fit(X, y)**. All of this in Figure 4.

```
>>> from ML import Perceptron
... pn = Perceptron(0.1, 10)
... pn.fit(X, y)
```

Figure 4: Importing the Perceptron Class from the *ML* library, creating an instance of the Perceptron Class, and then training it on the data set.

Lastly, we use the **plot_decision_regions()** function from the *ML* library, by using **from ML import plot_decision_regions**. We then will call the function directly by using **plot_decision_regions(X,y,pn,0.02)** and giving it the same X and y vector data sets, using the object we created of the Perceptron Class and giving a resolution of the graph it will generate. This is shown below.

```
>>> from ML import plot_decision_regions
... plot_decision_regions(X,y,pn,0.02)
```

Figure 5: Importing the plotting function that shows the region between the data set and the line the Perceptron Class found.

In Figure 6, we show the resulting output from the **plot_decision_regions()** function, which to reiterate only works on a feature set of two, any feature set past two, will result in it failing.

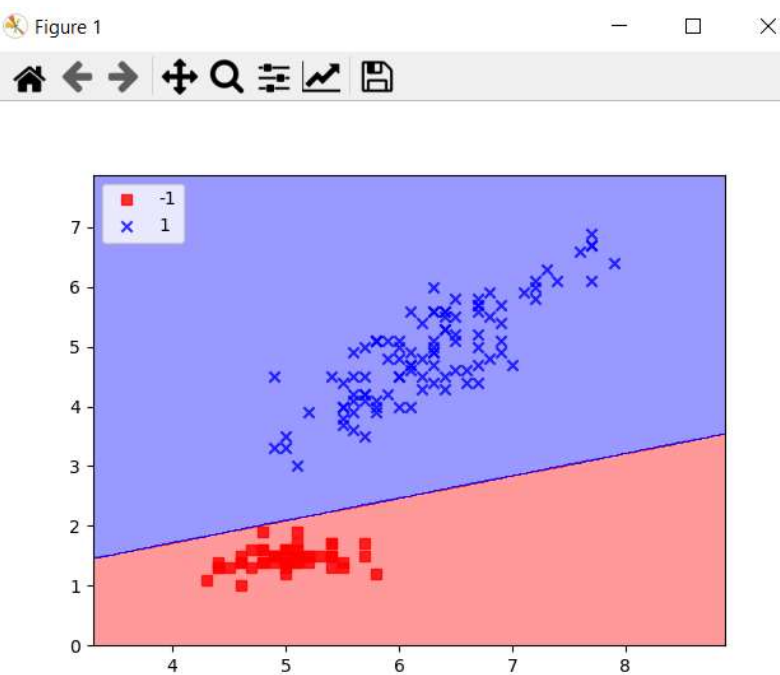


Figure 6: Resulting figure from the *plot_decision_regions* function.

The user may also add in the title and axis labels by using `plt.xlabel('sepal length [cm]')` and `plt.ylabel('petal length [cm]')` followed by `plt.title('Petal Length vs Sepal Length')`.

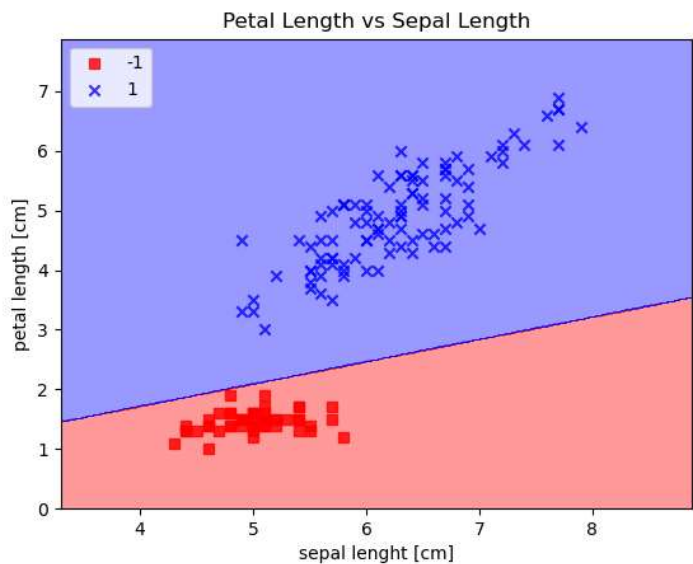


Figure 7: Final plot of the dataset and the Perceptron output.

To see how long it took the Perceptron to learn the dataset, we can simply use **pn.errors** this returns the error list as shown below.

```
>>> pn.errors
[2, 2, 3, 2, 1, 0, 0]
```

Figure 8: Showing how long it took for the Perceptron to learn the dataset.

Lastly, we will look at the weights of the Perceptron as it will be important later when we run the three-feature data set. We use **pn.weight**.

```
>>> pn.weight
array([-0.4 , -0.68,  1.82])
```

Figure 9: Showing the bias and weights used by the Perceptron to learn the dataset.

Three-feature set

We will now run a three-feature dataset the Perceptron, unfortunately we cannot use our **polt_desision_regions()** on any dataset that has more than one feature set. For this reason, everything will be done using the command line. We will start off with setting up our libraires and creating our target vector from the *Iris* dataset.

```
>>> import pandas as pd, numpy as np, matplotlib.pyplot as plt
... df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data', header=None)
... y = df.iloc[0:150, 4].values
... y = np.where(y == 'Iris-setosa', -1, 1)
```

Figure 10: Setting up the beginning of the three-feature.

Now here is the only difference in the setup between the two-feature set and the three-feature set: **X = df.iloc[0:150, [0,1,2]].values**. We add an extra column from the dataset to our training vector.

```
>>> X = df.iloc[0:150, [0,1,2]].values
```

Figure 11: Creating the three-feature training set vector.

Like last time we will need to import the Perceptron Class from the ML library and assign an object to the class. We will need to specify the training rate and number of iterations.

```
>>> from ML import Perceptron
... pn = Perceptron(0.1, 10)
... pn.fit(X, y)
```

Figure 12: Importing and creating the Perceptron Class.

Now the biggest difference is going to be found in the error list as well as the weight vector. We will check the Perceptron's error list as well as the weight vector. For each extra feature, the weight vector will have an extra weight added to its total length.

```
>>> pn.errors
[2, 2, 2, 2, 1, 0, 0]
>>> pn.weight
array([-0.2 , -0.3 , -1.24,  1.74])
```

Figure 13: Checking the amount of errors as well as the weights of the Perceptron Class on the three-feature data set.

Known Bugs and Errors

The known bug is found within the **plot_decision_regions()** function, which if it is given a three-feature data set it will have an indexing error that is off by one. Due to time constraints, we could not fix this problem, but hopefully by the next program this issue will be completed. In Figure 14, we show the resulting error from using the three-feature example mentioned above in the **plot_decision_regions()** function.

```
>>> from ML import plot_decision_regions
>>> plot_decision_regions(X,y,pn,0.02)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "C:\Users\7380314\PycharmProjects\MLstart\ML.py", line 83, in plot_decision_regions
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
  File "C:\Users\7380314\PycharmProjects\MLstart\ML.py", line 65, in predict
    return np.where(self.net_input(X) >= 0.0, 1, -1)
  File "C:\Users\7380314\PycharmProjects\MLstart\ML.py", line 61, in net_input
    return np.dot(X, self.weight[1:]) + self.weight[0]
  File "<__array_function__ internals>", line 5, in dot
ValueError: shapes (61600,2) and (3,) not aligned: 2 (dim 1) != 3 (dim 0)
```

Figure 14: Error of using the *plot_decision_regions* function not working with a three-feature data set.

Another “bug” or “error” that the user could run into is if the user does not let the Perceptron run enough iterations to learn the data set. This is not an error of the Perceptron; it is just a child and needs more time to master the dataset. The example below will simply create a Perceptron that runs fewer than the three-feature data set, but with only four iterations.

```
>>> pn1.errors
[2, 2, 2, 2]
>>> pn.errors
[2, 2, 2, 2, 1, 0, 0]
>>> pn1.weight
array([ 0.  ,  0.72, -0.54,  2.02])
>>> pn.weight
array([-0.2 , -0.3 , -1.24,  1.74])
```

Figure 15: The differences of two Perceptron Classes on the same data set, but **pn1** is not allowed to iterate enough through the data set enough to get the correct answer.

Potential Bug

Lastly, a potential error is for the **main.py** file not to work, but we believe it will work. We believe that the *PyCharm* IDE does not work well with the *matplotlib* unless it runs directly from the *python* console built in the IDE. This should however run correctly in other IDE's or from a *python* console. As a back up the **ML_test.txt** should be spaced correctly for the user to copy and paste the code to run the tests mentioned in this report. It is possible that our IDE just closes the plot extremely fast.

Again, this is the first we have used *python*, but to run the main file, all the user should have to type is **main()** and it should run the tests. If this does not work then please copy and paste from the **ML_test.txt** file, which hopefully be formatted to work with the *python* spacing.

Appendices

A Source Code

A.1 ML.py

```
# ML.py
import numpy as np
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt
from matplotlib.axes._axes import _log as matplotlib_axes_logger

class Perceptron(object):
    def __init__(self, rate=0.01, niter=10):
        self.rate = rate # learning rate
        self.niter = niter # number of iterations

    def fit(self, X, y):
        # Fit training data X : Training vectors, X.shape : [#samples, #features] y :
        # Target values, y.shape : [#samples]"""

        # determines if we need to stop more training cycles because we have learned
        # all we can
        convergence = 0

        # weights: create a weights array of right size and initialize
        self.weight = np.zeros(1 + X.shape[1])

        # Number of misclassifications, creates an array to hold the number of
        # misclassifications
        self.errors = []

        # main loop to fit the data to the labels
        for i in range(self.niter):
            # set iteration error to zero
            error = 0

            # loop over all the objects in X and corresponding y element
            for xi, target in zip(X, y):
                # calculate the needed (delta_w) update from previous step
                # delta_w = rate * (target - prediction current object)
                delta_w = self.rate * (target - self.predict(xi))

                # calculate what the current object will add to the weight Why not
                # self.weight[1:] += rate * (target - predict) * xi?
                self.weight[1:] += delta_w * xi

                # set the bias to be the current delta_w why not do self.weight[0] +=
                # rate*(target - predict)?
                self.weight[0] += delta_w

                # increase the iteration error if delta_w != 0
                error += int(delta_w != 0.0)
```

```

        # Update the misclassification array with # of errors in iteration
        self.errors.append(error)

        # add in check if we have reached 0 error
        if self.errors[i] == 0:
            # print("Convergence: ", convergence) #testing only
            convergence = convergence + 1
            # print("Convergence: ", convergence) #testing only
            if convergence == 2:
                return self
        # return self
        return self

    def net_input(self, X):
        """Calculate net input"""
        # return the return the dot product: X.w + bias
        return np.dot(X, self.weight[1:]) + self.weight[0]

    def predict(self, X):
        # Return class label after unit step
        return np.where(self.net_input(X) >= 0.0, 1, -1)

def plot_decision_regions(X, y, classifier, resolution=0.02):
    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    # create a np array
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # I considered using on of the predefined color gradient, but I got too flustered,
    so
    # Thanks to "Max Kleiner"on stackoverflow.com who recommended this work around,
    which lowers the error level
    # https://stackoverflow.com/questions/55109716/c-argument-looks-like-a-single-
    numeric-rgb-or-rgba-sequence
    matplotlib.axes_logger.setLevel('ERROR')

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution), np.arange(x2_min,
x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)

    # creates the the shaded regions of the data
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)

    # set min and max values of the given data set
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot class samples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1], alpha=0.8, c=cmap(idx),

```

```
marker=markers[idx], label=cl)

# add in the legend, so the used knows what values are being compared
plt.legend(loc='upper left')
```

A.2 main.py

```
import pandas as pd, numpy as np, matplotlib.pyplot as plt
from ML import Perceptron
from ML import plot_decision_regions

def main():
    df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data', header=None)
    print("Creating a two-feature data set")
    y = df.iloc[0:100, 4].values
    y = np.where(y == 'Iris-setosa', -1, 1)
    X = df.iloc[0:100, [0, 2]].values
    print("Creating Perceptron")
    pn = Perceptron(0.1, 10)
    print("Perceptron created")
    pn.fit(X, y)
    print("Perceptron fitted")
    print("Error List")
    print(pn.errors)
    print("Weight vector")
    print(pn.weight)
    print("Using plot_decision_regions function")
    plot_decision_regions(X, y, pn, 0.02)
    plt.xlabel('sepal length [cm]')
    plt.ylabel('petal length [cm]')
    plt.title('Petal Length vs Sepal Length')

    print("Creating a three-feature data set")
    y = df.iloc[0:150, 4].values
    y = np.where(y == 'Iris-setosa', -1, 1)
    X = df.iloc[0:150, [0, 1, 2]].values
    print("Creating Perceptron")
    pn1 = Perceptron(0.1, 10)
    print("Perceptron created")
    pn1.fit(X, y)
    print("Perceptron fitted")
    print("Error List")
    print(pn1.errors)
    print("Weight vector")
    print(pn1.weight)

    print("Creating a perceptron that does not have enough iterations to learn the
three-feature data set")
    pn2 = Perceptron(0.1, 4)
    print("Perceptron created")
    pn2.fit(X, y)
    print("Perceptron fitted")
    print("Error List")
    print(pn2.errors)
```

```

    print("Weight vector")
    print(pn2.weight)

if __name__ == "__main__":
    main()

```

A.3 ML_1_test.text

ML assingment 1

```

import pandas as pd, numpy as np, matplotlib.pyplot as plt
from ML import Perceptron
from ML import plot_decision_regions
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data', header=None)
print("Creating a two-feature data set")
y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', -1, 1)
X = df.iloc[0:100, [0, 2]].values
print("Creating Perceptron")
pn = Perceptron(0.1, 10)
print("Perceptron created")
pn.fit(X, y)
print("Perceptron fitted")
print("Error List")
print(pn.errors)
print("Weight vector")
print(pn.weight)
print("Using plot_decision_regions function")
plot_decision_regions(X, y, pn, 0.02)
plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.title('Petal Length vs Sepal Length')

print("Creating a three-feature data set")
y = df.iloc[0:150, 4].values
y = np.where(y == 'Iris-setosa', -1, 1)
X = df.iloc[0:150, [0,1,2]].values
print("Creating Perceptron")
pn1 = Perceptron(0.1, 10)
print("Perceptron created")
pn1.fit(X, y)
print("Perceptron fitted")
print("Error List")
print(pn1.errors)
print("Weight vector")
print(pn1.weight)

```

```
print("Creating a perceptron that does not have enough iterations to learn the three-feature data set")
pn2 = Perceptron(0.1, 4)
print("Perceptron created")
pn2.fit(X, y)
print("Perceptron fitted")
print("Error List")
print(pn2.errors)
print("Weight vector")
print(pn2.weight)
```